LOTOS Symbolic Semantics in Maude*

Alberto Verdejo

Technical Report 122-02
Dpto. Sistemas Informáticos y Programación
Universidad Complutense de Madrid. Spain

January 2002

*Research supported by CICYT project *Desarrollo Formal de Sistemas Basados en Agentes Móviles* (TIC2000–0701–C02–01).

Abstract

We present a formal tool where LOTOS specifications without restrictions in their data types can be executed. The reflective feature of rewriting logic and the metalanguage capabilities of Maude make it possible to implement the whole tool in the same semantic framework, and have allowed us to implement the LOTOS semantics and to build an entire environment with parsing, pretty printing, and input/output processing of LOTOS specifications.
Keywords: LOTOS, symbolic semantics, rewriting logic, Maude, meta-language.

Contents

1 Introduction

The formal description technique LOTOS [11] was developed within ISO for the formal specification of open distributed systems. Its behaviour description part is based on process algebras, borrowing ideas from CCS [15] and CSP [10], and the mechanism to define and to deal with data types is based on ACT ONE [8].[1] LOTOS became an international standard (IS-8807) in 1989. Since its standardization, LOTOS has been used to describe hundreds of systems, and most of this success is due to the existence of tools where specifications can be executed, compared, and analyzed.

The standard defines LOTOS semantics by means of labelled transition systems, where each data variable is instantiated by every possible value. That is the reason why most of the tools ignore or restrict the use of data types. Calder and Shankland [3] have defined a *symbolic* semantics for LOTOS which gives meaning to symbolic, or data parameterised processes (see below) and avoids infinite branching.

In this paper we use rewriting logic [13, 14] and Maude [6] to implement a formal tool based on this symbolic semantics where LOTOS specifications without restrictions in their data types can be executed.

1.1 LOTOS symbolic semantics

The implementation given here is entirely based on the work presented in [2, 3]. A symbolic semantics for LOTOS is given by associating a symbolic transition system with each LOTOS behaviour expression P. Following [9], Calder and Shankland define *symbolic transition systems* (STS) as transition systems which separate the data from process behaviour by making the data symbolic. STS are labelled transition systems with variables, both in states and transitions, and conditions which determine the validity of a transition.

Definition 1 *(Symbolic Transition Systems)*
A symbolic transition system *consists of:*

- *A (nonempty) set of states. Each state T is associated with a set of free variables, denoted $fv(T)$.*
- *A distinguished initial state, T_0.*
- *A set of transitions written as $T \xrightarrow{b \quad \alpha} T'$, where α is a simple or structured event and b is a Boolean expression, such that $fv(T') \subseteq fv(T) \cup fv(\alpha)$, $fv(b) \subseteq fv(T) \cup fv(\alpha)$ and $\#(fv(\alpha) - fv(T)) \leq 1$.*

In the symbolic semantics, *open* behaviour expressions label states (for example, $h\,!\,x$ **; stop**), and transitions offer variables, under some conditions; these conditions determine the set of values which may be substituted for variables.

In [3] the intuition and key features of this semantics are presented, together with axioms and rules of transition for each LOTOS operator. We will see them, together with their representation in Maude, in Section 2.2.

1.2 Rewriting logic and Maude

Rewriting logic was introduced by Meseguer [13] as a unified model of concurrency in which several well-known models of concurrent systems can be represented in a common framework. Since then much work has been done on the use of rewriting logic as a logical and semantic framework [12], in which many different logics, models of computation, and a wide range of languages, including formal specification languages like LOTOS, can be represented, can be given a precise semantics, and can be executed. Maude, a high-level language and high-performance system supporting both equational and rewriting logic computation [6], should be viewed as a *metalanguage* [5] in which the syntax and semantics of all these models and languages can be formally defined, and in which entire *environments* for such languages can be built (including parsers, execution environments, pretty

[1] The union of the behaviour and data type description parts is known as Full LOTOS. We use in this paper the term LOTOS to refer to the whole language.

printing, and input/output). We will see how an environment of this kind has been built for LOTOS in Section 3.

Reflection is the main feature to achieve these powerful metalanguage functionalities. Rewriting logic is reflective [4], that is, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$, and we then have the following equivalence:

$$\mathcal{R} \vdash t \longrightarrow t' \;\; \Leftrightarrow \;\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`, where Maude terms are reified as elements of a data type `Term`, Maude modules are reified as terms in a data type `Module`, the process of reducing a term to normal form is reified by a function `meta-reduce`, and the process of applying a rule of a system module to a subject term is reified by a function `meta-apply` [6]. These basic operations can be combined to build *strategies* that control the process of rewriting.

The capability of Maude to be used as a metalanguage has been applied to extend Maude itself. Full Maude [7] is an extension of the Maude language with notation for object-oriented programming, parameterized modules, views (for module instantiation), and module expressions. Full Maude and its execution environment have been completely implemented in (Core) Maude. We use part of this implementation to build our tool for LOTOS, as we will see in Section 3.

If we want to use rewriting logic as a semantic framework and Maude as a metalanguage to *implement* our language, the first thing we have to do is to represent the language $\mathcal{L}$ in question in rewriting logic by a mapping of the form

$$\Phi : \mathcal{L} \longrightarrow \mathit{RWLogic}.$$

In our present case, the map Φ is essentially an identity map, preserving the original structure of the formulas, and mirroring each semantic rule by a corresponding rewrite rule. We will show this representation in Sections 2.1 and 2.2.

Then, we would like to *execute* that representation. That is when reflection and the module `META-LEVEL` are at work. We will show in Sections 2.3 and 2.4 how a search strategy can be defined and how it can be used to execute the LOTOS symbolic semantics. This part of the paper is based on previous work [18, 19], although with important modifications to adapt it to the LOTOS semantics characteristics.

But if we want to build a usable formal tool we need more. We have to build an *environment* for it, including not only the execution aspect just described, but parsing, pretty printing, and input/output. We will describe how it has been done for LOTOS in Section 3. This is the main contribution of this paper.

2 LOTOS symbolic semantics in Maude

We presented with all the details an implementation of the operational semantics of CCS in [18]. Now we present the general ideas for this kind of representations, the problems we found and our solutions, and how our approach has been used to represent the LOTOS symbolic semantics.

The general idea for implementing in rewriting logic an operational semantics, is to translate each semantic rule into a rewrite rule where either the premises are rewritten to the conclusion, or the conclusion is rewritten to the premises [12]. We adopt the second approach, because we want to be able to prove in a bottom-up goal-directed way that a given transition is valid.

In the present case, symbolic transitions are represented as terms of sort `Transition`, and the semantic rules are translated into rewrite rules where the representation of the conclusion is rewritten to the set of representations of the premises. In this way, we start with a transition to be proved valid and work backwards using the rewriting process, maintaining a set of transitions that have to be fulfilled in order to prove the correctness of the initial transition. This transition can be rewritten to the empty set if and only if it is a valid transition in the LOTOS symbolic semantics.

But we found two problems while working with this approach in the current version of Maude. The first one is that sometimes new variables appear in the premises which are not in the conclusion. Rules of this kind, with new variables in the righthand side, cannot be directly used by the Maude default interpreter; they can only be used at the metalevel using a strategy to instantiate the extra variables.

Another problem is that sometimes several rules can be applied to rewrite a transition, but, in general, not all the possibilities lead to an empty set. So we have to deal with the whole tree of possible rewrites of a transition, searching if one of the branches leads to the empty set.

2.1 LOTOS syntax

In this section we are going to introduce the LOTOS syntax. It is defined in the Maude functional module `LOTOS-SYNTAX`. We use the predefined quoted identifiers to build LOTOS variable, sort, and gate identifiers. Booleans are the only predefined data type. LOTOS syntax is extended in a user-definable way when ACT ONE data types specifications are used. Values of these data types will extend the type `DataExp` below. We will see how it is done in Section 3.

```
(fmod LOTOS-SYNTAX is
  protecting QID .

  *** identifiers contructors
  sorts VarId SortId GateId .

  op V : Qid -> VarId .
  op S : Qid -> SortId .
  op G : Qid -> GateId .

  sort DataExp .

  subsort VarId < DataExp .

  subsort Bool < DataExp .
```

The basic processes in LOTOS are inaction (deadlock), and successful termination possibly with data values. The exit paramenter can be a value or a variable over a set of values. For simplicity, as it is done in [3], we will assume that only one parameter can occur at an `exit` behaviour expression.

```
  sort BehaviourExp .

  op stop : -> BehaviourExp .
  op exit : -> BehaviourExp .
  op exit‘(_‘) : ExitParam -> BehaviourExp .

  sort ExitParam .
  subsort DataExp < ExitParam .

  op any_ : SortId -> ExitParam .
```

Actions occur at gates, and may or may not have data offers associated with them. For simplicity we will also assume that only one event offer can occur at an action. Actions may also be subject to *selection predicates*. The special action `i` is the (unobservable) internal event.

```
  sorts SimpleAction StrucAction Action Offer SelecPred IdDecl .

  subsort GateId < SimpleAction .
  subsorts SimpleAction StrucAction < Action .

  op i : -> SimpleAction .
  op __ : GateId Offer -> StrucAction [prec 30] .
```

```
  op !_ : DataExp -> Offer [prec 25] .
  op ?_ : IdDecl -> Offer [prec 25] .
  op _:_ : VarId SortId -> IdDecl [prec 20] .

  op _`[_`] : SimpleAction SelecPred -> Action [prec 30] .
  op _`[_`] : StrucAction SelecPred -> Action [prec 30] .

  subsort Bool < SelecPred .
```

Actions and processes are combined using the following operators:

```
  *** action prefixing
  op _;_ : Action BehaviourExp -> BehaviourExp [prec 35] .

  *** choice
  op _`[`]_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .
  op choice_`[`]_ : IdDecl BehaviourExp -> BehaviourExp [prec 40] .

  *** parallelism
  sort GateIdList .
  subsort GateId < GateIdList .

  op _`,_ : GateIdList GateIdList -> GateIdList [prec 35] .


  op _|`[_`]|_ : BehaviourExp GateIdList BehaviourExp ->
                 BehaviourExp [prec 40] .
  op _||_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .
  op _|||_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .

  *** disable
  op _`[>_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .

  *** guard
  op `[_`]->_ : SelecPred BehaviourExp -> BehaviourExp [prec 40] .

  *** hide
  op hide_in_ : GateIdList BehaviourExp -> BehaviourExp [prec 40] .

endfm)
```

2.2 LOTOS symbolic semantics

In order to represent in Maude the LOTOS symbolic semantics we follow the ideas presented in [17].

First, we define the elements of a symbolic transition, that is, events and transition conditions.

```
(mod LOTOS-SYMBOLIC-SEMANTICS is
  pr LOTOS-SYNTAX .

  sorts SimpleEv StructEv Event TransCond .

  subsort SimpleAction < SimpleEv .
  op delta : -> GateId .

  op __ : GateId DataExp -> StructEv .

  subsorts SimpleEv StructEv < Event .

  *** Trace of events
```

```
sort Trace .
subsort Event < Trace .
op nil : -> Trace .
op __ : Trace Trace -> Trace [assoc id: nil prec 35 gather(e E)] .

*** transition conditions

subsort Bool < TransCond .
var B : Bool .
subsort SelecPred < TransCond .
op _=_ : DataExp DataExp -> TransCond [prec 25] .


op _/\_ : TransCond TransCond -> TransCond [assoc comm] .
var b : TransCond .

eq b /\ true = b .

var A : SimpleAction .
vars E E1 E2 : DataExp .
vars NEW1 NEW2 : Qid .

vars g g' : GateId .
var SP : SelecPred .
vars x y z : VarId .
var S : SortId .
var GIL : GateIdList .
```

We define a sort of `Judgement`s to represent the basic elements of a semantic rule. Common judgements will be LOTOS transitions, but we will see below other kinds of judgements.

In [17], we worked with sets of judgements. In this case, for efficiency reasons, that is, in order to avoid multiple matching modulo commutativity, we work with sequences of judgements. This means that judgements will be ordered, and, as we will see below, they will be proved from left to right, that is, if the first judgement of a sequence cannot be reduced to the empty sequence, then we know that the whole sequence cannot be reduced, and we can abandon it.

```
sorts Judgement JudgementSeq .

op emptyJS : -> JudgementSeq .
subsort Judgement < JudgementSeq .
op __ : JudgementSeq JudgementSeq -> JudgementSeq [assoc id: emptyJS prec 60] .

var J : Judgement .
vars JS JS' : JudgementSeq .

sort Configuration .

op `{`{_`}`} : JudgementSeq -> Configuration .
op `{`{_|_`}`} : JudgementSeq JudgementSeq -> Configuration .
```

In order to solve the problem of new variables in the righthand side of a rewrite rule, we use *explicit metavariables*. Metavariables are needed as transition conditions, events, and behaviour expressions.

In the case of metavariables as transition conditions, we define a new sort `MetaVarTransCond`, an operator for building new metavariables from quoted identifiers, and a new sort `TransCond?` of "possible transition conditions."

```
sorts MetaVarTransCond TransCond? .
subsorts MetaVarTransCond TransCond < TransCond? .

op ?`(_`)b : Qid -> MetaVarTransCond .
```

We also need a judgement for representing the binding of a metavariable to a concrete value, and rules to propagate this binding to the rest of judgements. In [18] we used auxiliary functions to perform the substitution of values for metavariables. We defined them by means of equations that distinguish cases based on the constructors of the terms where the substitution is being applied. We cannot do the same now, because we do not know the syntax of data expressions. We use the operator `_[[_/_]]` to represent a *syntactic substitution*. Its behaviour is defined at the metalevel, as we will see in Section 2.4. A judgement for representing the equality of transition conditions is also needed. It is eliminated when both metavariables have been bound to the same concrete value.

```
op `[_:=_`] : MetaVarTransCond TransCond? -> Judgement .
op _`[`[_/_`]`] : JudgementSeq TransCond MetaVarTransCond -> JudgementSeq .

vars b b' : TransCond .
var ?b : MetaVarTransCond .
var b? : TransCond? .

rl [bind] : {{ [?b := b] JS }} => {{ JS [[ b / ?b ]] }} .

rl [bind] : {{ [?b := b] JS | JS' }} =>
            {{ (JS [[ b / ?b ]]) | [?b := b] JS' }} .

op `[_==_`] : TransCond? TransCond? -> Judgement .

rl [equal] : [b == b] => emptyJS .
```

We do the same for metavariables as events, and metavariables as behaviour expressions.

```
  *** metavariables as events

sort MetaVarEvent Event? .
subsorts MetaVarEvent Event < Event? .
op ?`(_`)a : Qid -> MetaVarEvent .
op `[_:=_`] : MetaVarEvent Event? -> Judgement .
op _`[`[_/_`]`] : JudgementSeq Event MetaVarEvent -> JudgementSeq .

op __ : Event? VarId -> Event? .

vars a a' : Event .
var ?a : MetaVarEvent .
var a? : Event? .

rl [bind] : {{ [?a := a] JS }} => {{ JS [[ a / ?a ]] }} .

rl [bind] : {{ [?a := a] JS | JS' }} =>
            {{ (JS [[ a / ?a ]]) | [?a := a] JS' }} .

op `[_==_`] : Event? Event? -> Judgement .

rl [equal] : [a == a] => emptyJS .

  *** metavariables as BehaviourExp

sort MetaVarBehExp BehExp? .
subsorts MetaVarBehExp BehaviourExp < BehExp? .
op ?`(_`)P : Qid -> MetaVarBehExp .
op `[_:=_`] : MetaVarBehExp BehExp?  -> Judgement .
op _`[`[_/_`]`] : JudgementSeq BehaviourExp MetaVarBehExp -> JudgementSeq .

vars P P' P1 P2 : BehaviourExp .
var ?P : MetaVarBehExp .
```

```
var P? : BehExp? .

rl [bind] : {{ [?P := P] JS }} => {{ JS [[ P / ?P ]] }} .

rl [bind] : {{ [?P := P] JS | JS' }} =>
            {{ (JS [[ P / ?P ]]) | [?P := P] JS' }} .

op `[_==_`] : BehExp? BehExp? -> Judgement .

rl [equal] : [P == P] => emptyJS .
```

Now we can define an operator for building symbolic transitions.

```
sort Transition .
subsort Transition < Judgement .

op _--_--_-->_ : BehaviourExp TransCond? Event? BehExp? ->
                 Transition [prec 50] .
```

Before defining the semantic rules, we define several functions used by the semantics. In the semantics, a set **new-var** of fresh variable names is assumed. For building new variable names, we use the same idea as for building new metavariables as explained in [18]. In the semantic rules we will use `new-var(NEW1)`, where `NEW1` is a new variable in the righthand side of the rewrite rule, and which is substituted by new quoted identifiers when the rules are applied (at the metalevel by the search strategy).

```
op new-var : Qid -> VarId .

var Q : Qid .
eq new-var(Q) = V(conc('z@, Q)) .
```

A (data) substitution is written as $[z/x]$ where z is substituted for x. The syntax and semantics of the substitution operators is as follows:

```
sort Substitution .

op `[_/_`] : VarId VarId? -> Substitution .
op `[`] : -> Substitution .

op __ : BehExp? Substitution -> BehExp? .
op __ : TransCond? Substitution -> TransCond? .
op __ : Event? Substitution -> Event? .

eq P? [] = P? .
eq b? [] = b? .
eq a? [] = a? .

*** syntactic substitution (defined at the metalevel)

op _`[`[_/_`]`] : BehaviourExp VarId VarId -> BehaviourExp .
op _`[`[_/_`]`] : TransCond VarId VarId -> TransCond .
op _`[`[_/_`]`] : Event VarId VarId -> Event .

eq P [ y / x ] = P [[ y / x ]] .
eq b [ y / x ] = b [[ y / x ]] .
eq a [ y / x ] = a [[ y / x ]] .

op _`[`[_/_`]`] : BehaviourExp DataExp VarId -> BehaviourExp .
op _`[`[_/_`]`] : TransCond DataExp VarId -> TransCond .
op _`[`[_/_`]`] : Event DataExp VarId -> Event .
```

The function **name**() : Act $\cup$ $\{\delta, \mathbf{i}\} \rightarrow G \cup \{\delta, \mathbf{i}\}$ extracts the gate name from a structured event.

```
op name : Event? -> Event? .

eq name(i) = i .
eq name(g) = g .
eq name(g E) = g .
```

A predicate to know if a given gate identifier appears in a given gate identifier list is also used by the semantics. We extend the operator to metavariables as events.

```
op _in_ : Event? GateIdList -> Bool .

eq i in GIL = false .
eq g in g' = (g == g') .
eq g in (g' , GIL) = (g == g') or (g in GIL) .
```

We also define some auxiliary functions useful for the representation of the semantics. We define an operation `hide` that given an event `a` and a gate identifier list, returns the internal action `i` if the name of event `a` is in the given list, and returns `a` in other case.

```
op hide : Event? GateIdList -> Event? .

ceq hide(a, GIL) = i if name(a) in GIL .
ceq hide(a, GIL) = a if not(name(a) in GIL) .
```

A predicate for knowing if an event is structured, and an operation to extract the variable name of a structured event are also defined.

```
op structured : Event? -> Bool .

eq structured(i) = false .
eq structured(g) = false .
eq structured(g E) = true .

sort VarId? . subsort VarId < VarId? .
op var : Event? -> VarId? .

eq var(g x) = x .

sort DataExp? . subsort DataExp < DataExp? .

op value : Event? -> DataExp? .

eq value(g E) = E .

op _=_ : DataExp? DataExp? -> TransCond? .
```

In the semantics definition a function *vars* is used to obtain the variables occurring in a behaviour expression. Since a behaviour may have data expressions, and these are built by means of a user-definable syntax, we cannot define at this level an operation to extract the variables in a behaviour. We declare an operation `vars` which is defined by means of another operation `vars@metalevel` which will be defined at the metalevel (see Section 2.4), where the behaviour (including data expressions) will be metarepresented as a `Term` and we will be able to traverse it extracting (metarepresented) LOTOS variables.

```
sorts VarSet .
subsort VarId < VarSet .
```

```
op mt : -> VarSet .
op _U_ : VarSet VarSet -> VarSet [assoc comm id: mt] .

eq x U x = x . *** idempotency

op _in_ : VarId VarSet -> Bool .

var VS : VarSet .

eq x in mt = false .
eq x in (y U VS) = (x == y) or (x in VS) .

op vars : BehExp? -> VarSet .
op vars@metalevel : BehaviourExp -> VarSet .

eq vars(P) = vars@metalevel(P) .
```

The following operation returns the substitution needed in the semantics rules for the parallel operator, depending on the given action.

```
op subsPar : Event? VarSet VarId -> Substitution .

eq subsPar(A, VS, z) = [] .
eq subsPar(g E, VS, z) = if (E : VarId) then
                          (if (E in VS) then [ z / var(g E) ]
                           else [] fi)
                         else [] fi .
```

Other kinds of auxiliary judgements are also used in the semantics representation: a judgement for representing the fact that two actions have to be different, and a judgement enclosing a Boolean predicate (which may have metavariables).

```
*** other judgements...

op `[_=/=_`] : Event? Event? -> Judgement .

crl [dist] : [a =/= a'] => emptyJS  if a =/= a' .


op <_> : Bool -> Judgement .

rl [bool] : < true > => emptyJS .
```

Some of the LOTOS syntax operators have to be overloaded, allowing metavariables.

```
*** overloaded LOTOS operators

op hide_in_ : GateIdList BehExp? -> BehExp? [prec 40] .
op _`[>_ : BehExp? BehExp? -> BehExp? [prec 40] .
op _|`[_`]|_ : BehExp? GateIdList BehExp? -> BehExp? [prec 40] .

op _/\_ : TransCond? TransCond? -> TransCond? [assoc comm] .
```

Now, we can represent the LOTOS symbolic semantics rules in Maude. For each LOTOS operator, we present the semantic rules and their representation as rewrite rules.[2]

In [17], we wrote for each operator several rules depending on the places where metavariables appear. In this case, when we present the rules of the operators we always assume metavariables

[2]Using the fact that text beginning with --- is a comment in Maude, rules are displayed in such a way as to emphasize the correspondence with the usual presentation in textbooks, although in this case the conclusion is above the horizontal line.

everywhere, that is, as transition condition, event and resulting behaviour expression. At the end of this section we present how other kinds of transitions (without metavariables everywhere) are translated.

```
*************************************************************
*** rules of transition for symbolic semantics
*** These rules are the ones presented in "A Symbolic
*** Semantics and Bisimulation for Full LOTOS" (October 2000)
```

prefix axioms

$$a; P \xrightarrow{\text{tt} \quad a} P$$

$$g\ d; P \xrightarrow{\text{tt} \quad gE'} P$$

$$g\ d[SP]; P \xrightarrow{SP \qquad gE'} P$$

$$E' = \begin{cases} E & \text{if} \quad d = !E \\ x & \text{if} \quad d = ?x{:}S \end{cases}$$

```
*** prefix axioms

rl [sym] :
      A ; P -- ?b -- ?a --> ?P
  => --------------------------------
        [ ?b := true ]
        [ ?a := A ]
        [ ?P := P ] .

rl [sym] :
      A [SP] ; P -- ?b -- ?a --> ?P
  => --------------------------------
        [ ?b := SP ]
        [ ?a := A ]
        [ ?P := P ] .

rl [sym] :
      g ! E ; P -- ?b -- ?a --> ?P
  => --------------------------------
        [ ?b := true ]
        [ ?a := g E ]
        [ ?P := P ] .

rl [sym] :
      g ! E [ SP ] ; P -- ?b -- ?a --> ?P
  => --------------------------------
        [ ?b := SP ]
        [ ?a := g E ]
        [ ?P := P ] .

rl [sym] :
      g ? x : S ; P -- ?b -- ?a --> ?P
  => --------------------------------
        [ ?b := true ]
        [ ?a := g x ]
        [ ?P := P ] .

rl [sym] :
      g ? x : S [SP] ; P -- ?b -- ?a --> ?P
  => ---------------------------------------
```

```
        [ ?b := SP ]
        [ ?a := g x ]
        [ ?P := P ] .
```

exit axioms

$$\text{exit} \xrightarrow{\text{tt} \quad \delta} \text{stop}$$

$$\text{exit(ep)} \xrightarrow{\text{tt} \quad \delta E'} \text{stop}$$

$$E' = \begin{cases} E & \text{if ep } = E \\ z & \text{if ep } = \textbf{any } S \quad \text{where } z \in \textbf{new-var}. \end{cases}$$

```
*** exit axioms

rl [sym] :
      exit -- ?b -- ?a --> ?P
  => --------------------------------
        [ ?b := true ]
        [ ?a := delta ]
        [ ?P := stop ] .

rl [sym] :
      exit(E) -- ?b -- ?a --> ?P
  => --------------------------------
        [ ?b := true ]
        [ ?a := delta E ]
        [ ?P := stop ] .

rl [sym] :
      exit(any S) -- ?b -- ?a --> ?P
  => --------------------------------
        [ ?b := true ]
        [ ?a := delta new-var(NEW1) ]
        [ ?P := stop ] .
```

choice range rules

$$\frac{P[g_i/g] \xrightarrow{b \quad \alpha} P'}{\mathit{choice}\ g\ \mathit{in}\ [g_1, \ldots, g_n]\ [\,]\ P \xrightarrow{b \quad \alpha} P'}$$

for each $g_i \in \{g_1, \ldots, g_n\}$

$$\frac{P \xrightarrow{b \quad \alpha} P'}{\mathit{choice}\ x : S\ [\,]\ P \xrightarrow{b \quad \alpha} P'}$$

```
*** choice range rules

rl [sym] :
      choice x : S [] P -- ?b -- ?a --> ?P
  => ---------------------------------------
            P -- ?b -- ?a --> ?P .
```

hide rules

$$\frac{P \xrightarrow{b \quad \alpha} P'}{\mathit{hide}\ g_1, \ldots, g_n\ \mathit{in}\ P \xrightarrow{b \quad i} \mathit{hide}\ g_1, \ldots, g_n\ \mathit{in}\ P'}$$

if $\textbf{name}(\alpha) \in \{g_1, \ldots, g_n\}$

$$\frac{P \xrightarrow{b \quad \alpha} P'}{hide\ g_1, \ldots, g_n\ in\ P \xrightarrow{b \quad \alpha} hide\ g_1, \ldots, g_n\ in\ P'}$$

if $\mathbf{name}(\alpha) \notin \{g_1, \ldots, g_n\}$

```
*** hide rules

rl [sym] :
      hide GIL in P -- ?b -- ?a --> ?P
  => --------------------------------
       P -- ?b -- ?(NEW1)a --> ?(NEW1)P
       [ ?a := hide(?(NEW1)a, GIL) ]
       [ ?P := hide GIL in ?(NEW1)P ] .
```

disable rules

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 [> P_2 \xrightarrow{b \quad \alpha} P_1' [> P_2}$$

if $\mathbf{name}(\alpha) \neq \delta$

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 [> P_2 \xrightarrow{b \quad \alpha} P_1'}$$

if $\mathbf{name}(\alpha) = \delta$

$$\frac{P_2 \xrightarrow{b \quad \alpha} P_2'}{P_1 [> P_2 \xrightarrow{b \quad \alpha} P_2'}$$

```
*** disable rules

rl [sym] :
      P1 [> P2 -- ?b -- ?a --> ?P
  => --------------------------------
       P1 -- ?b -- ?a --> ?(NEW1)P
       [ name(?a) =/= delta ]
       [ ?P := ?(NEW1)P [> P2 ] .

rl [sym] :
      P1 [> P2 -- ?b -- ?a --> ?P
  => --------------------------------
       P1 -- ?b -- ?a --> ?P
       [ name(?a) == delta ] .

rl [sym] :
      P1 [> P2 -- ?b -- ?a --> ?P
  => --------------------------------
       P2 -- ?b -- ?a --> ?P .
```

general parallelism rules (not synchronising)

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 | [g_1, \ldots, g_n] | P_2 \xrightarrow{b\sigma \quad \alpha\sigma} P_1'\sigma | [g_1, \ldots, g_n] | P_2}$$

$\mathbf{name}(\alpha) \notin \{g_1, \ldots, g_n, \delta\}$

$$\sigma = \begin{cases} [z/x] & \text{if } \alpha = gx \text{ and } x \in vars(P_2) \quad \text{where } z \in \mathbf{new\text{-}var}. \\ [\,] & \text{otherwise} \end{cases}$$

Similarly for P_2.

```
*** general parallelism rules (not synchronising)

rl [sym] :
          P1 |[ GIL ]| P2 -- ?b -- ?a --> ?P
  => ----------------------------------------
       P1 -- ?(NEW1)b -- ?(NEW1)a --> ?(NEW1)P
       < not( name(?(NEW1)a) in (GIL, delta) ) >
       [ ?b := ?(NEW1)b subsPar(?(NEW1)a, vars(P2), new-var(NEW1)) ]
       [ ?a := ?(NEW1)a subsPar(?(NEW1)a, vars(P2), new-var(NEW1)) ]
       [ ?P := ( ?(NEW1)P subsPar(?(NEW1)a, vars(P2), new-var(NEW1)))
               |[ GIL ]|
                 P2 ] .

rl [sym] :
          P1 |[ GIL ]| P2 -- ?b -- ?a --> ?P
  => ----------------------------------------
       P2 -- ?(NEW1)b -- ?(NEW1)a --> ?(NEW1)P
       < not( name(?(NEW1)a) in (GIL, delta) ) >
       [ ?b := ?(NEW1)b subsPar(?(NEW1)a, vars(P1), new-var(NEW1)) ]
       [ ?a := ?(NEW1)a subsPar(?(NEW1)a, vars(P1), new-var(NEW1)) ]
       [ ?P :=  P1
             |[ GIL ]|
              (?(NEW1)P subsPar(?(NEW1)a, vars(P1), new-var(NEW1))) ] .
```

general parallelism rules (synchronising)

$$\frac{P_1 \xrightarrow{b_1 \quad g} P_1' \qquad P_2 \xrightarrow{b_2 \quad g} P_2'}{P_1 | [g_1, \ldots, g_n] | P_2 \xrightarrow{b_1 \wedge b_2 \quad g} P_1' | [g_1, \ldots, g_n] | P_2'}$$

where $g \in \{g_1, \ldots, g_n, \delta\}$

$$\frac{P_1 \xrightarrow{b_1 \quad gE_1} P_1' \qquad P_2 \xrightarrow{b_2 \quad gE_2} P_2'}{P_1 | [g_1, \ldots, g_n] | P_2 \xrightarrow{b_1 \wedge b_2 \wedge E_1 = E_2 \quad gE_1} P_1' | [g_1, \ldots, g_n] | P_2'}$$

when $vars(b_1 \cup E_1) \cap vars(b_2 \cup E_2) = \emptyset$.

```
*** general parallelism rules (synchronising without values)

rl [sym] :
          P1 |[ GIL ]| P2 -- ?b -- ?a --> ?P
  => ----------------------------------------
       P1 -- ?(NEW1)b -- ?a --> ?(NEW1)P
       P2 -- ?(NEW2)b -- ?a --> ?(NEW2)P
       < name(?a) in (GIL, delta) >
       < not(structured(?a)) >
       [ ?b := ?(NEW1)b /\ ?(NEW2)b ]
       [ ?P := ?(NEW1)P |[ GIL ]| ?(NEW2)P ] .


*** general parallelism rules (synchronising with values)

rl [sym] :
          P1 |[ GIL ]| P2 -- ?b -- ?a --> ?P
  => ----------------------------------------
       P1 -- ?(NEW1)b -- ?(NEW1)a --> ?(NEW1)P
       P2 -- ?(NEW2)b -- ?(NEW2)a --> ?(NEW2)P
       [ name(?(NEW1)a) == name(?(NEW2)a) ]
       < name(?(NEW1)a) in (GIL, delta) >
       < structured(?(NEW1)a) >
       < structured(?(NEW2)a) >
       [ ?b := ?(NEW1)b /\ ?(NEW2)b /\
```

```
            (value(?(NEW1)a) = value(?(NEW2)a)) ]
        [ ?a := ?(NEW1)a ]
        [ ?P := ?(NEW1)P |[ GIL ]| ?(NEW2)P ] .
```

choice rules

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 \,[\,]\, P_2 \xrightarrow{b \quad \alpha} P_1'}$$

$$\frac{P_2 \xrightarrow{b \quad \alpha} P_2'}{P_1 \,[\,]\, P_2 \xrightarrow{b \quad \alpha} P_2'}$$

```
*** choice rules

rl [sym] :
      P1 [] P2 -- ?b -- ?a --> ?P
  => --------------------------------
        P1 -- ?b -- ?a --> ?P .

rl [sym] :
      P1 [] P2 -- ?b -- ?a --> ?P
  => --------------------------------
        P2 -- ?b -- ?a --> ?P .
```

guard rule

$$\frac{P \xrightarrow{b \quad \alpha} P'}{([SP]\ \texttt{->}\ P) \xrightarrow{b \wedge SP \quad \alpha} P'}$$

```
*** guard rule

rl [sym] :
      [ SP ] -> P -- ?b -- ?a --> ?P
  => --------------------------------
        P -- ?(NEW1)b -- ?a --> ?P
        [ ?b := ?(NEW1)b /\ SP ] .
```

Other kinds of starting judgements are reduced to the above ones as follows:

```
rl [sym] :
       P -- b -- a --> P'
  => ---------------------------------------
       P -- ?(NEW1)b -- ?(NEW1)a --> ?(NEW1)P
       [ ?(NEW1)b == b ]
       [ ?(NEW1)a == a ]
       [ ?(NEW1)P == P' ] .

rl [sym] :
       P -- b -- a --> ?P
  => ---------------------------------------
       P -- ?(NEW1)b -- ?(NEW1)a --> ?P
       [ ?(NEW1)b == b ]
       [ ?(NEW1)a == a ] .

rl [sym] :
       P -- b -- ?a --> P'
  => ---------------------------------------
```

```
        P -- ?(NEW1)b -- ?a --> ?(NEW1)P
        [ ?(NEW1)b == b ]
        [ ?(NEW1)P == P' ] .

  rl [sym] :
        P -- b -- ?a --> ?P
    => ----------------------------------------
        P -- ?(NEW1)b -- ?a --> ?P
        [ ?(NEW1)b == b ] .

  rl [sym] :
        P -- ?b -- a --> P'
    => ----------------------------------------
        P -- ?b -- ?(NEW1)a --> ?(NEW1)P
        [ ?(NEW1)a == a ]
        [ ?(NEW1)P == P' ] .

  rl [sym] :
        P -- ?b -- a --> ?P
    => ----------------------------------------
        P -- ?b -- ?(NEW1)a --> ?P
        [ ?(NEW1)a == a ] .

  rl [sym] :
        P -- ?b -- ?a --> P'
    => ----------------------------------------
        P -- ?b -- ?a --> ?(NEW1)P
        [ ?(NEW1)P == P' ] .
```

The bindings of the form `x = E` that may appear in a transition condition when two behaviours synchronize, are not propagated to the resulting behaviour by the symbolic semantics. Actually, the value of any variable can be figured out by tracing through the conditions, traversing the symbolic transition system. However, the LOTOS tool we define below (and which uses the semantics previously defined) will propagate these bindings in order to show in a more readable way the possible transitions of a behaviour. We can define the propagation of the bindings in a transition condition at this level.

```
  ceq (E1 = E2) = (E2 = E1) if E2 : VarId and not(E1 : VarId) .

  op apply-subst : TransCond BehaviourExp -> BehaviourExp .

  eq apply-subst(B, P) = P .
  eq apply-subst(E1 = E2, P) =
     if (E1 : VarId) and not(E2 : VarId) then
       P [[ E2 / E1 ]]
     else if (E2 : VarId) and not(E1 : VarId) then
            P [[ E1 / E2 ]]
          else P
          fi
     fi .
  eq apply-subst(b /\ b', P ) =
       apply-subst(b, apply-subst(b',P)) .

endm)
```

2.3 Searching in the tree of rewrites

We developed in [18] a general enough search strategy to control the rewriting of a term and the search in the tree of all possible rewrites of a term. The strategy also helps to deal with metavariables. We

extended META-LEVEL with operations that generate all the one-step rewrites of a term, and operations that search in a depth-first way the tree of possible rewrites looking for the empty set of transitions.

We have modified the strategy in [18] for efficiency reasons. When we are trying to rewrite a sequence of judgements, we only try to rewrite its *first* judgement. As we said above, the judgements are ordered, and we have to test that all of them can be rewritten to the empty sequence. So if the first judgement cannot be rewritten we do not need to rewrite the rest of judgements, and we can drop all the sequence.

The search strategy has the following parameters: a constant (MOD) representing the module, a constant (labels) representing the list of labels of rewrite rules to be applied, a constant (num-metavar) representing the number of different metavariables that can appear in the same rule, and a constant (operators) representing the list of operators whose arguments have to be rewritten when looking for all the rewrites of a term, that is, the operators whose arguments rewriting makes sense.

```
(fmod PARAMS is
  including META-LEVEL .
  op MOD : -> Module .
  op labels : -> QidList .
  op num-metavar : -> MachineInt .
  op operators : -> QidList .
endfm)
```

This module is included by the SEARCH module, and the constants will be defined by the module which uses the search strategy, for example, the module LOTOS-TOOLS below.

Since we are defining a strategy to search a tree of possible rewrites, we need a notion of search goal. For the strategy to be general enough, we assume that the module MOD has a predicate ok (defined at the object level), which defines when a term is one of the terms we are looking for, that is, it denotes a solution.

```
(fmod SEARCH is
  inc PARAMS .
  sort TermSet .
  subsort Term < TermSet .

  vars F C S V OP OP' : Qid .
  vars T T' X Y : Term .
  vars TL TL' Before After : TermList .
  vars M N : MachineInt .
  var L : Qid .
  vars LS OPS : QidList .
  var SB : Substitution .
  var TS : TermSet .
  var PS : PairSeq .
```

The predicate isInQidList returns true if the given operator identifier (of sort Qid) appears in the given list.

```
  op isInQidList : Qid QidList -> Bool .

  eq isInQidList(OP, nil) = false .
  eq isInQidList(OP, (OP' OPS)) =
     if (OP == OP') then true else isInQidList(OP, OPS) fi .
```

The strategy controls the possible rewrites of a term by means of the metalevel function meta-apply. The evaluation of meta-apply(MOD,T,L,S,N) applies (discarding the first N successful matches) a rule of module MOD with label L, partially instantiated with substitution S, to the term T (at the top level). It returns the resulting fully reduced term and the representation of the match used in the reduction, { T', S' }, as a term of sort ResultPair built by means of the operator {_,_}.

In Section **sec:semantics** we saw the necessity of instantiating the new variables in the righthand side of a rewrite rule in order to create new metavariables. We have to provide a substitution in such a way that the rules are always applied without new variables in the righthand side.

The operation `new-var` receives an integer and returns the metarepresentation of a quoted identifier representing that number. For example, the evaluation of `new-var(5)` returns the metarepresentation `{''5}'Qid` of term `'5`.

The operation `createSubst` receives as first argument the number of metavariables (new variables in the righthand side of a rewrite rule) that have to be substituted and, as second argument, the greatest number used to build new quoted identifiers. It returns the substitution that substitutes the new variables by new identifiers. For example,

```
createSubst(3, 7) = 'NEW1@Qid <- { ''8 }'Qid ;
                    'NEW2@Qid <- { ''9 }'Qid ;
                    'NEW3@Qid <- { ''10 }'Qid
```

```
*** creation of new metavariables

op new-var : MachineInt -> Term .
eq new-var(N) = {conc('' , index(' , N))}'Qid .

op createSubst : MachineInt MachineInt -> Substitution .

eq createSubst(0, M) = none .
ceq createSubst(N, M) = (createSubst(_-_(N,1), M) ;
                         (conc('NEW , conc(index(' , N), '@Qid))
                          <- new-var(M + N))) if N =/= 0 .
```

We define now a new operation `meta-apply'` which receives the greatest number `M` used to substitute variables in `T` and uses new numbers to create new (metarepresented) identifiers.

```
op extTerm : ResultPair -> Term .
op extSubst : ResultPair -> Substitution .

eq extTerm({T, SB}) = T .
eq extSubst({T, SB}) = SB .

op meta-apply' : Term Qid MachineInt MachineInt -> Term .

eq meta-apply'(T, L, N, M) =
    extTerm(meta-apply(MOD, T, L, createSubst(num-metavar, M), N)) .
```

The operation `meta-apply'` returns *one* of the possible one-step rewrites at the top level of a given term. In [17], we defined an operation `allRew` that returns *all* the possible *one-step sequential* rewrites [13] of a given term `T` by using rewrite rules with labels in the list `labels`. In this case, we do not want to do a blind search of all possible rewrites, but we introduce knowledge about the terms we are rewriting. We only rewrite the arguments of a term if the main operator is in the list `operators` (in the LOTOS case these operators are `{{_}}`, `{{_|_}}`, and `__`). And we only rewrite the *first* argument. When the operator is `{{_}}` or `{{_|_}}` only the first argument can be rewritten. As we said above, the judgements are ordered, and we have to test that all of them can be rewritten to the empty sequence. So if the first judgement cannot be rewritten we do not need to rewrite the rest of judgements, and we can drop all the sequence.

The third argument of `allRew` represents the greatest number `M` used to substitute new variables in `T`.

The operations needed to find all the possible rewrites, and their definitions, are as follows:

```
op allRew : Term QidList MachineInt  -> TermList .
op topRew : Term Qid MachineInt MachineInt -> TermList .
op lowerRew : Term Qid MachineInt -> TermList .
```

```
op rewArguments : Qid TermList TermList Qid MachineInt -> TermList .
op rebuild : Qid TermList TermList TermList -> TermList .

eq allRew(T, nil, M) = ~ .
eq allRew(T, L LS, M) = topRew(T, L, 0, M) ,
                        lowerRew(T, L, M) , allRew(T, LS, M) .
```

The evaluation of `topRew(T, L, N, M)` returns all the possible one-step rewritings at the top level of term `T` applying rule `L`, discarding the first `N` matches, and using numbers from `M+1` to create identifiers for new variables.

```
eq topRew(T, L, N, M) =
    if meta-apply'(T, L, N, M) == error* then ~
    else (meta-apply'(T, L, N, M) , topRew(T, L, N + 1, M))
    fi .
```

The evaluation of `lowerRew(T,L,M)` returns all the interesting rewrites of the subterms of term `T` applying rule `L`, and using numbers from `M+1` to create identifiers for new variables.

```
eq lowerRew({C}S, L, M) = ~ .
eq lowerRew(OP[TL], L, M) =
   if isInQidList(OP, operators) then rewArguments(OP, ~, TL, L, M)
   else ~ fi .

eq rewArguments(OP, Before, T, L, M) =
    rebuild(OP, Before, allRew(T, L, M), ~) .
eq rewArguments(OP, Before, (T, After), L, M) =
*** only the first argument is rewritten
    rebuild(OP, Before, allRew(T, L, M), After) .
```

The evaluation of `rebuild(OP, B, TL, A)` returns all the terms of the form `OP[B, T, A]` where `T` is a term in the term list `TL`. These built terms are metareduced before being returned.

```
eq rebuild(OP, Before, ~, After) = ~ .
eq rebuild(OP, Before, T, After) = meta-reduce(MOD, OP[Before, T, After]) .
eq rebuild(OP, Before, (T, TL), After) =
    meta-reduce(MOD, OP[Before, T, After]) ,
    rebuild(OP, Before, TL, After) .
```

Now we can define a strategy to search in the (conceptual) tree of all possible rewrites of a term `T` for a term that satisfies the `ok` predicate. Each node of the search tree is a pair, whose first component is a term and whose second component is a number representing the greatest number used as identifier for new variables in the process of rewriting the term. The tree nodes that have been generated but not yet been checked are maintained in a sequence.

```
sorts Pair PairSeq .
subsort Pair < PairSeq .

op <_`,_> : Term MachineInt -> Pair .
op nil : -> PairSeq .
op _|_ : PairSeq PairSeq -> PairSeq [assoc id: nil] .
```

We need an operation to build these pairs from the list of terms produced by `allRew`:

```
op buildPairs : TermList MachineInt -> PairSeq .

eq buildPairs(~, N) = nil .
eq buildPairs(T, N) = < T , N > .
eq buildPairs((T, TL), N) = < T , N > | buildPairs(TL, N) .
```

The operation `rewDepth` starts the search by calling the operation `rewDepth'` with the root of the search tree. `rewDepth'` returns the first solution found in a depth-first way as a singleton pair sequence `< T , N >`. If there is no solution, the empty sequence term is returned.

```
  op rewDepth : Term -> PairSeq .
  op rewDepth' : PairSeq -> PairSeq .

  eq rewDepth(T) = rewDepth'(< meta-reduce(MOD, T), 0 >) .

  eq rewDepth'(nil) = nil .
  eq rewDepth'(< T , N > | PS) =
      if meta-reduce(MOD, 'ok[T]) == {'solution}'Answer then < T , N >
      else (if meta-reduce(MOD, 'ok[T]) == {'no-solution}'Answer then
                     rewDepth'(PS)
            else rewDepth'(buildPairs(allRew(T, labels, N),
                                      (N + num-metavar)) | PS )
            fi)
      fi .
```

The operation `rewDepth` only returns one solution, but we can modify it in order to get all the solutions, that is, in order to explore (in a depth-first way) the whole tree of rewrites finding *all* the nodes that satisfy the predicate `ok`. The operation `allSol` returns a sequence of pairs whose first components are the terms representing all the solutions. This operation can be called with one argument of sort `Term`, which is the metarepresented term with which we want to start the search. There is another version of this operation with two arguments. The first argument `T` is as before, and the second one is a number which represents the greatest number used as identifier for new variables in a previous search that finished in `T`.

```
  *** All solutions

  op allSol : Term -> PairSeq .
  op allSol : Term MachineInt -> PairSeq .

  eq allSol(T) = allSol(T, 0) .
  eq allSol(T, N) = allSolDepth(< meta-reduce(MOD,T) , N >) .

  op allSolDepth : PairSeq -> PairSeq .

  eq allSolDepth(nil) = nil .
  eq allSolDepth(< T , N > | PS) =
    if meta-reduce(MOD, 'ok[T]) == {'solution}'Answer then
                (< T , N > | allSolDepth(PS))
    else (if meta-reduce(MOD, 'ok[T]) == {'no-solution}'Answer then
                    allSolDepth(PS)
          else
            allSolDepth(buildPairs(allRew(T, labels, N), N + num-metavar) | PS)
          fi)
    fi .

endfm)
```

2.4 How the semantics and search strategy are used

Now we can instantiate the search strategy with the LOTOS semantics rules in order to make it executable. First, we specify a module `LOTOS-OK` extending the LOTOS syntax and semantic rules by defining the predicate `ok` that states when a configuration is a solution. A configuration denotes a solution when it is the empty sequence of judgements, representing that the sequence of judgements at the beginning is provable by means of the semantic rules.

```
(mod LOTOS-OK is
  protecting LOTOS-SYMBOLIC-SEMANTICS .

  sort Answer .
  ops solution no-solution maybe-sol : -> Answer .

  op ok : Configuration -> Answer .

  vars JS JS' : JudgementSeq .

  eq ok({{ emptyJS }}) = solution .
  ceq ok({{ JS }}) = maybe-sol if JS =/= emptyJS .

  eq ok({{ emptyJS | JS' }}) = solution .
  ceq ok({{ JS | JS'}}) = maybe-sol if JS =/= emptyJS .
```

We define a sort `PossibleTrans` of possible transitions (of an initial behaviour expression). These operators are used by the operations below to represent at the object level the solutions returned by the search strategy.

```
  sort PossibleTrans .

  op |--_--_-->_ : TransCond Event BehaviourExp -> PossibleTrans [prec 50] .

  op nil : -> PossibleTrans .
  op _&_ : PossibleTrans PossibleTrans -> PossibleTrans [assoc id: nil prec 60] .

endm)
```

The following module *instantiates* part of the constants in module `PARAMS`. `MOD` is the only constant which is left undefined. It should have the metarepresented module with the syntax and semantics of LOTOS. But, at this point, the syntax is not complete because the syntax for data values is not defined yet. It will be defined by the user in ACT ONE. At that moment, we will be able to build a module that includes the functional modules which are the translation of the ACT ONE modules and the LOTOS syntax and semantics. We will metarepresent it and build a module with an equation that equals the constant `MOD` to this metarepresented module (see Section 3.5).

```
(fmod LOTOS-TOOLS is
  including SEARCH .

  eq labels = ( 'bind 'dist 'equal 'bool 'sym) .
  eq num-metavar = 2 .
  eq operators =  (''{''{_'}'} ''{''{_|_'}'}  '__ ) .

  vars F C S V OP OP' : Qid .
  var A T T' BS MV X Y : Term .
  vars MVS TS AS : TermSet .
  vars TL TL' : TermList .
  var N : MachineInt .
  var PS : PairSeq .
```

We had left two things uncompleted, waiting for their completion at the metalevel: syntactic substitution and extraction of variables of a behaviour expression. The reason why we cannot do them when defining the semantics is the same in both cases: the presence of data expressions with user-definable syntax. At the metalevel a fixed, known syntax is used to metarepresent terms, so we are able to define both operations.

```
  *** syntactic substitution
```

```
op replace : Term Term Term -> Term .
op replaceList : TermList Term Term -> TermList .

ceq  '_`[`[_/_`]`] [ T, Y, X ] = replace(T, Y, X) if not(T : Qid) .

eq replace(T, Y, T) = Y .
ceq replace({C}S, Y, X) = {C}S if X =/= {C}S .

ceq replace(OP[TL], Y, X) =
    meta-reduce(MOD, OP[ replaceList(TL, Y, X) ]) if X =/= OP[TL] .


eq replaceList(T, Y, X) = replace(T, Y, X) .
eq replaceList((T, TL), Y, X) = replace(T, Y, X), replaceList(TL, Y, X) .


*** vars AT META-LEVEL

op vars@metalevel : TermList -> Term .
op vars@metalevel2 : TermList -> Term .

ceq 'vars@metalevel [ T ] = vars@metalevel(T) if not(T : Qid) .

eq vars@metalevel(T) = meta-reduce(MOD, vars@metalevel2(T)) .

eq vars@metalevel2({C}S) = {'mt}'VarSet .
eq vars@metalevel2(V) = {'mt}'VarSet .
eq vars@metalevel2( 'V[TL] ) = 'V[TL] .
ceq vars@metalevel2( F[TL] ) = vars@metalevel2(TL) if F =/= 'V .
eq vars@metalevel2((TL,TL')) = '_U_ [ vars@metalevel2(TL),
                                     vars@metalevel2(TL') ] .
```

The operation `filter-bind` receives the metarepresentation of a sequence of binding judgements and the metarepresentation of a metavariable and returns the term metarepresenting the concrete value to which that metavariable has been bound.

```
op filter-bind : TermList Term -> Term .

eq filter-bind( ('`[_:=_`] [T, T']) , MV) =
   if (T == MV) then  T'  else  error*  fi .

eq filter-bind( ('`[_:=_`] [T, T'], TL) , MV) =
   if (T == MV) then  T'  else  filter-bind(TL, MV)  fi .

eq filter-bind('__ [ TL ] , MV) = filter-bind(TL, MV) .
```

The operation `transitions` receives the metarepresentation of a LOTOS behaviour expression and returns a sequence with the metarepresentations of all its possible transitions as metarepresented terms of sort `PossibleTrans`, defined in the module `LOTOS-OK`. The transitions of process `P` are calculated by `allSol`, which searches in the tree with root the transition

```
P -- ?('b)b -- ?('a)a --> ?('P)P
```

As with `allSol`, operation `transitions` can be called with two arguments; in this case, the second one represents the greatest number used as identifier for new variables in a previous search that finished in the term given as first argument.

```
op transitions : Term -> PairSeq .
op transitions : Term MachineInt -> PairSeq .
```

```
  op get-transitions : PairSeq -> PairSeq .
  op get-trans : Term -> Term .

  eq transitions(T) = transitions(T, 0) .

  eq transitions(T, N) = apply-subst(get-transitions(allSol(
         ''{'{_|_'}'} [ '_--_--_-->_ [ T,
                                      '?'(_')b [ {''b}'Qid ] ,
                                      '?'(_')a [ {''a}'Qid ] ,
                                      '?'(_')P [ {''P}'Qid ]] ,
                        {'emptyJS}'JudgementSeq ], N))) .
```

The operation `get-transitions` receives the solutions found and it returns a list of possible transtions, one for each solution. It filters the bindings between metavariables and values that go with each solution, looking for the concrete metavariables `?('b)b`, `?('a)a`, and `?('P)P`.

```
  eq get-transitions( nil ) = nil .
  eq get-transitions( < ''{'{_|_'}'}[{'emptyJS}'JudgementSeq, BS], N > | PS)
                   = < get-trans(BS), N > | get-transitions(PS)  .

  eq get-trans(BS) = '|--_--_-->_ [
                        filter-bind(BS, '?'(_')b [ {''b}'Qid ]),
                        filter-bind(BS, '?'(_')a [ {''a}'Qid ]),
                        filter-bind(BS, '?'(_')P [ {''P}'Qid ]) ] .
```

The operation `apply-subst` modifies each possible transition propagating the bindings in the transition condition (if any) to the resulting behaviour expression.

```
  op apply-subst : PairSeq -> PairSeq .
  op apply-subst : Term -> Term .

  eq apply-subst(nil) = (nil).PairSeq .
  eq apply-subst(< T, N > | PS) =
       < apply-subst(T), N > | apply-subst(PS) .

  eq apply-subst('|--_--_-->_ [T, A, T']) =
       '|--_--_-->_ [T, A,
                     meta-reduce(MOD, 'apply-subst[T,T'])] .
```

We also define an operation `downSol` that transforms a sequence of pairs where the first component is a term metarepresenting a possible transition in module `LOTOS-OK`, into a term metarepresenting all the transitions in the sequence. This can be used together with the Full Maude command `down` to show at the object level (the LOTOS semantics level) the results of operation `transitions`.

```
  op downSol : PairSeq -> Term .

  eq downSol(nil) = {'nil}'PossibleTrans .
  eq downSol(< T, N >) = T .
  ceq downSol(< T, N > | PS) = '_&_[T, downSol(PS)] if PS =/= nil .

endfm)
```

A module like the following one should be used to define the constant `MOD`. It will be introduced to the Full Maude database of modules when a Full LOTOS specification is entered to our tool, although instead of `LOTOS-OK`, the metarepresented module will be `EXT-LOTOS-OK` which will include the extended syntax of user definable data types. We use the Full Maude `up` function [6, 7] to obtain the metarepresentation of a module or a term.

```
(fmod FULL-LOTOS is
  inc LOTOS-TOOLS .
  eq MOD = up(LOTOS-OK) .
endfm)
```

We can already use the `FULL-LOTOS` module, since it is executable. We can use the `transitions` operation to know the possible transtions of a LOTOS behaviour (although without data expressions). But it would be quite cumbersome because we should use metarepresented terms both for input and output from this operation.

As we saw in the introduction, an entire environment for LOTOS can be built by using the metalanguage features of Maude. We show how it can be done in the next section.

3 Building the LOTOS tool environment

Maude has the following metalanguage features for parsing, pretty printing, and input/output [6]:

- The *syntax definition* for the language $\mathcal{L}$ is accomplished by defining a data type $\texttt{Grammar}_{\mathcal{L}}$, which can be done with very flexible user-definable *mixfix* syntax, that can mirror the concrete syntax of $\mathcal{L}$. Particularities at the lexical level of $\mathcal{L}$ can be accommodated by user-definable *bubble sorts*, that tailor the adequate notions of token and identifier to the language in question. Bubbles correspond to pieces of a module in a language that can only be parsed once the grammar introduced by the signature of the module is available. This is specially important when $\mathcal{L}$ has user-definable syntax, as it is our case with ACT ONE.

- Parsing and pretty printing for $\mathcal{L}$ is accomplished by the `meta-parse` and `meta-pretty-print` functions in `META-LEVEL`. Function `meta-parse` receives as arguments the representation of a module M and the representation of a list of tokens and it returns the metarepresentation of the parsed term (a parse tree that may have bubbles) of that list of tokens for the signature of M. Function `meta-pretty-print` receives the representation of a module M and a term t, and it returns a list of quoted identifiers that encode the string of tokens produced by pretty printing t in the syntax given by M.

- Input/output of $\mathcal{L}$ specifications, and of commands for execution in $\mathcal{L}$ is accomplished by the predefined module `LOOP-MODE`, that provides a generic read-eval-print loop. This module has an operator `[_,_,_]` that can be seen as a persistent object with an input and output channel (the first and third arguments, respectively), and a state (given by its second argument). We have complete flexibility for defining this state. In Section 3.5 we will see how we extend the state used by Full Maude. When something is written in the Maude prompt enclosed in parentheses[3] it is placed in the first slot of the loop object, as a list of quoted identifiers. The output is handled in the reverse way, that is, the list of quoted identifiers placed in the third slot of the loop is printed on the terminal.

As we said, all these techniques have been used to extend Maude itself, in the implementation of Full Maude [7]. We use this implementation not only because we want to use the auxiliary functions defined there, as we will see in Section 3.2, but because we need the database of modules maintained by Full Maude. Full Maude maintains as the state of the loop object a database of modules entered to the system. It is needed to be able to execute commands in the modules, and to perform the module operations defined by Full Maude. And we need the database to be able to build new modules *on the fly*, when LOTOS specifications are entered to the system, as we will see in Section 3.5

3.1 The grammar of the LOTOS tool interface

We have to define first the signature (syntax) of Full LOTOS (ACT ONE and LOTOS) and the signature of the commands we are going to use in our tool to work with the entered specification.[4]

[3] As we have done with the modules in Section 2, since they will be introduced into Full Maude.

[4] There is an important separation between the signature used by the users to write their specifications and the abstract syntax we defined in Section 2.1.

The signature of ACT ONE and LOTOS is given in Appendix A.

Part of the syntax of Full LOTOS, due to the ACT ONE data types, is user-definable. Maude provides great flexibility to define this syntax thanks to its mixfix front-end and to the use of *bubbles* [7].

The following module, `LOTOS-TOOL-SIGN`, includes the ACT ONE and LOTOS signature and the commands of our tool.

```
fmod LOTOS-TOOL-SIGN is
  protecting LOTOS-SIGN .
  pr MACHINE-INT .

  sort LotosCommand .

  op L show process . : -> LotosCommand .
  op L show possible transitions . : -> LotosCommand .
  op L show possible transitions of_. : BehaviourExp -> LotosCommand .
  op L trans . : -> LotosCommand .
  op L trans of_. : BehaviourExp -> LotosCommand .
  op L cont_. : MachineInt -> LotosCommand .
  op L cont . :  -> LotosCommand .
  op L show state . : -> LotosCommand .

endfm
```

The first command is used to show the current process, that is, the behaviour expression used if we omit it in the rest of commands. The second and third commands are used to show the possible transitions (defined by the symbolic semantics) of the current or explicitly given process, that is, they start the execution of a process. The fourth command is used to continue the execution with one of the possible transitions, the one indicated in the argument of the command. `L cont` is a shorthand for `L cont 1`.

In order to parse some input using the built-in function `meta-parse`, we need to give the metarepresentation of the signature in which the input is going to be parsed. By including the module `LOTOS-TOOL-SIGN` in the metarepresented module `LOTOS-GRAMMAR` we get the metarepresentation of the signature. The `LOTOS-GRAMMAR` module will be used in calls to the `meta-parse` function in order to get the input parsed in this signature. Notice that from the call to `meta-parse` we will get a term representing the parse tree of the input (maybe with bubbles). This term will then be transformed into a term of an appropriate data type.

```
fmod META-LOTOS-TOOL-SIGN is
  inc META-LEVEL .
  pr META-FULL-MAUDE-SIGN .
  pr UNIT .

  op LOTOS-GRAMMAR : -> FModule .
  eq LOTOS-GRAMMAR
    = (fmod 'LOTOS-GRAMMAR is
        including 'QID-LIST .
        including 'LOTOS-TOOL-SIGN .
        sorts none .
        none
        op 'token : 'Qid -> 'Token
             [special(
               (id-hook('Bubble, '1 '1)
                op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)))] .
        op 'neTokenList : 'QidList -> 'NeTokenList
             [special(
               (id-hook('Bubble, '1 '-1 ''( ''))
                op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)
                id-hook('Exclude, '.)))] .
```

```
      op 'expBubble : 'QidList -> 'ExpBubble
           [special(
             (id-hook('Bubble, '1 '-1 ''( ''))
              op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
              op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)
              id-hook('Exclude, '. '! '=> '; 'any 'ofsort ''[ ''] )))] .
      op 'bubble : 'QidList -> 'Bubble
           [special(
             (id-hook('Bubble, '1 '-1 ''( ''))
              op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
              op-hook('qidBaseSymbol, '<Qids>, nil, 'Qid)
              id-hook('Exclude, '. '! '=> '; 'any 'ofsort ''[ ''] )))] .
      none
      none
      none
    endfm) .

endfm
```

3.2 ACT ONE modules translation

Instead of defining a datatype for representing ACT ONE modules in Maude and operations to transform the parse tree returned by `meta-parse` into a value of this datatype, we are going to use Maude functional modules to represent (internally) ACT ONE modules. Since Full Maude already has a function (`processUnit`) to transform a parse tree (maybe with bubbles) representing functional modules into functional modules, we have to define only a function `translate` that translates a parse tree representing an ACT ONE module into a parse tree representing a functional module.

$$\texttt{QidList} \xrightarrow{\texttt{meta-parse}} \texttt{PreModule}_{\text{ACT ONE}} \xrightarrow{\texttt{translate}} \texttt{PreModule} \xrightarrow{\texttt{processUnit}} \texttt{FModule}$$

```
fmod ACTONE-TRANSLATION is
  pr META-LEVEL .
  pr DATABASE-HANDLING .
  pr DECL-META-PRETTY-PRINT .
```

The operation `translate` has three arguments: the name of the LOTOS specification entered, the parse tree returned by `meta-parse` representing a list of ACT ONE datatype specifications, and a Full Maude database of modules. It returns this database modified by introducing a functional module for each ACT ONE datatype and one more module (with the name of the specification) which includes the introduced modules. This last module will be used to represent all the data types. We only show the translation of sort declarations.

```
  op translate : Term Term Database -> Database .
  op translateType : Term Term Database -> Database .
  op translateTypeList : Term Term Database TermList -> Database .
  op translateImportList : Term -> TermList .
  op translateDeclList : Term -> Term .
  op translateOperDeclList : Term -> Term .
  op translateSortNameList : Term -> Term .
  op translateVarEqDeclList : Term -> Term .
  op translateVariDeclList : Term -> Term .
  op translateEqDeclList : Term -> Term .

  op normalize : Term Term -> Term .

  vars T T' T'' T''' : Term .
  var F : Qid .
  var TL : TermList .
```

```
var DB : Database .

op typeName : Term -> Term .
eq typeName('type_is_endtype[T',T'']) = T' .
eq typeName('type_is__endtype[T',T'',T''']) = T' .

op include : TermList -> Term .
eq include(T) = 'including_.[T] .
eq include((T,TL)) = '__['including_.[T], include(TL)] .

eq translate(T, T', DB) = translateTypeList(T, T', DB, nilTermList) .

eq translateType(T, 'type_is_endtype[T',T''], DB) =
   processUnit('fmod_is_endfm[T',
                '__['including_.['token[{''LOTOS-SYNTAX}'Qid]],
                    translateDeclList(T'')]], DB) .

eq translateType(T, 'type_is__endtype[T',T'',T'''], DB) =
   processUnit('fmod_is_endfm[T',
                normalize(include(('token[{''LOTOS-SYNTAX}'Qid],
                          translateImportList(T'''))),
                    translateDeclList(T''''))], DB) .

eq translateTypeList(T, 'type_is_endtype[T',T''], DB, TL) =
   processUnit('fmod_is_endfm[T, include((TL,T''))],
               translateType(T,'type_is_endtype[T',T''], DB)) .
eq translateTypeList(T, 'type_is__endtype[T',T'',T'''], DB, TL) =
   processUnit('fmod_is_endfm[T, include((TL,T''))],
               translateType(T,'type_is__endtype[T',T'',T'''], DB)) .

eq translateTypeList(T, '__[T',T''], DB, TL) =
   translateTypeList(T, T'',
                     translateType(T,T', DB), (TL,typeName(T'))) .

eq translateImportList('token[T]) = 'token[T] .
eq translateImportList('_`,_[T,T']) = (translateImportList(T),
                                       translateImportList(T')) .
```

When an ACT ONE sort declaration for sort `T` is found it is not only translated into a Maude sort declaration for sort `T`, but we also have to declare type `T` as a subsort of sort `DataExp` (since values of the declared type could be used in a behaviour expression to be communicated) and the sort of LOTOS variables `VarId` has to be declared as a subsort of type `T` (since LOTOS variables could be used to build values of this type).

```
eq translateDeclList('sorts_['token[T]]) =
  '__['sort_.['sortToken[T]],
      '__['subsort_.['_<_['sortToken[T],'sortToken[{''DataExp}'Qid]]],
          'subsort_.['_<_['sortToken[{''VarId}'Qid],'sortToken[T]]]]] .

eq translateDeclList('opns_[T]) = translateOperDeclList(T) .

eq translateDeclList('eqns_[T]) = translateVarEqDeclList(T) .

eq translateDeclList('__[T,T']) =
   normalize(translateDeclList(T), translateDeclList(T')) .

eq translateOperDeclList('__[T,T']) =
   '__[translateOperDeclList(T), translateOperDeclList(T')] .

eq translateOperDeclList('_:`->_['token[T],'token[T']]) =
```

```
      'op_:'->_.['token[T],'sortToken[T']] .

  eq translateOperDeclList('_:_->_['token[T],T','token[T'']]) =
      'op_:_->_.['token[T],translateSortNameList(T'),'sortToken[T'']] .

  eq translateOperDeclList('_:'->_['_',_[T,T'],'token[T'']]) =
      'ops_:'->_.['neTokenList['__[elems('_',_[T,T'])]],'sortToken[T'']] .

  eq translateOperDeclList('_:_->_['_',_[T,T'],T'',token[T''']]) =
      'ops_:_->_.['neTokenList['__[elems('_',_[T,T'])]],
                  translateSortNameList(T''),'sortToken[T''']] .

  eq translateSortNameList('token[T]) = 'sortToken[T] .
  eq translateSortNameList('_',_[T,T']) =
  '__[translateSortNameList(T),translateSortNameList(T')] .

  eq translateVarEqDeclList('__[T,T']) =
     normalize(translateVarEqDeclList(T), translateVarEqDeclList(T')) .

  eq translateVarEqDeclList('forall_[T]) = translateVariDeclList(T) .

  eq translateVariDeclList('_',_[T,T']) =
     '__[translateVariDeclList(T), translateVariDeclList(T')] .

  eq translateVariDeclList('_:_['token[T],'token[T']]) =
      'var_:_.['neTokenList[T],'sortToken[T']] .

  op elems : Term -> TermList .

  eq elems('token[T]) = T .
  eq elems('_',_['token[T],T']) = T , elems(T') .

  eq translateVariDeclList('_:_['_',_[T,T'],'token[T'']]) =
      'vars_:_.['neTokenList['__[elems('_',_[T,T'])]],'sortToken[T'']] .

  eq translateVarEqDeclList('ofsort__[T,T']) = translateEqDeclList(T') .

  eq translateEqDeclList('__[T,T']) =
     '__[translateEqDeclList(T), translateEqDeclList(T')] .

  eq translateEqDeclList('_=_;[T,T']) = 'eq_=_.[T,T'] .
  eq translateEqDeclList('_=>_=_;[T,T',T'']) = 'ceq_=_if_.[T',T'',T] .

  ceq normalize(F[TL],T) = '__[F[TL],T] if F =/= '__ .
  eq normalize('__[T,T'],T'') = '__[T,normalize(T',T'')] .
endfm
```

The following is an example of how an ACT ONE specification is translated into a functional module. The ACT ONE specification

```
specification SPECIFICATION
type Naturals is
  sorts
    nat
  opns
    0 : -> nat
    s : nat -> nat
endtype
type Extended-Naturals is Naturals
  opns
    _+_ : nat , nat -> nat
```

```
  eqns
    forall x, y : nat
    ofsort nat
      0 + x = x ;
      s(x) + y = s(x + y) ;
endtype
```

is translated into

```
fmod Naturals is
  including LOTOS-SYNTAX .
  including BOOL .
  sorts nat .
  subsort VarId < nat .
  subsort nat < DataExp .
  op 0 : -> nat .
  op s : nat -> nat .
endfm

fmod Extended-Naturals is
  including LOTOS-SYNTAX .
  including Naturals .
  including BOOL .
  op _+_ : nat nat -> nat .
  var x : nat .
  var y : nat .
  eq s ( x ) + y = s ( x + y ) .
  eq 0 + x = x .
endfm

fmod SPECIFICATION is
  including Naturals .
  including BOOL .
  including Extended-Naturals .
endfm
```

3.3 LOTOS input processing

When LOTOS behaviour expressions are introduced, either as part of a whole specification or in a tool command, they have to be transformed into elements of the data type `BehaviourExp` in module `LOTOS-SYNTAX` (Section 2.1). The parse tree returned by `meta-parse` with module `LOTOS-GRAMMAR` may have bubbles (where data expressions may appear) that have to be parsed again using the user-defined syntax. This syntax can be found in the functional module that includes all the modules which are the translations of the types defined in ACT ONE (see module `SPECIFICATION` above). Moreover, the behaviour itself can define new syntax, since it can declare new LOTOS variables by means of `?` offers, and these variables may appear in expressions. For example, when processing the behaviour expression

```
g ? x : nat ; h ! s(x) + s(0) ; stop
```

the data expression `s(x) + s(0)` should be parsed using the signature in module `SPECIFICATION` extended with variable `x` of sort `nat`.

```
fmod LOTOS-PARSING is
  pr META-LEVEL .
  pr UNIT-DECL-PARSING .

  vars QI F C S V Q Q1 Q2 : Qid .
  var QIL : QidList .
  vars N : MachineInt .
```

```
var M : Module .
vars T T' T'' PS N' T1 T2 T3 : Term .
var TL : TermList .

sort VarSet .
op _:_ : Qid Qid -> VarSet .
op mt : -> VarSet .
op __ : VarSet VarSet -> VarSet [assoc comm id: mt] .

op varsToVarDeclSet : VarSet -> EVarDeclSet .

eq varsToVarDeclSet(mt) = none .
eq varsToVarDeclSet((V : S) VS) = (var V : S .)
                                  varsToVarDeclSet(VS) .
sort TermVars .
op <_`,_> : Term VarSet -> TermVars .

op term : TermVars -> Term .
op vars : TermVars -> VarSet .

var VS : VarSet .

eq term(< T, VS >) = T .
eq vars(< T, VS >) = VS .
```

We use the operation `parseProcess` to make this translation. It receives as arguments the term returned by `meta-parse` (representing a behaviour expression), the metarepresented module with the data types syntax (module `SPEC` above) and the set of free variables that may appear in the behaviour expression. It returns a behaviour expression without bubbles. It uses the operation `parseAction` that, besides the term metarepresenting the given action (without bubbles), returns the variables declared in the action (if any).

```
op parseProcess : Term Module VarSet -> Term .
op parseExitParam : Term Module VarSet -> Term .
op parseDataExp : Term Module VarSet -> Term .
op parseAction : Term Module VarSet -> TermVars .
op parseOffer : Term Module VarSet -> TermVars .
op parseGateIdList : Term -> Term .

op varsMaudeToLotos : TermList -> Term .

eq varsMaudeToLotos(V) = 'V[up(V)] .
eq varsMaudeToLotos({C}S) = {C}S .
eq varsMaudeToLotos(F[TL]) = F[varsMaudeToLotos(TL)] .
eq varsMaudeToLotos((T, TL)) =
   (varsMaudeToLotos(T), varsMaudeToLotos(TL)) .

eq parseExitParam('expBubble[T], M, VS) =
       parseDataExp('expBubble[T], M, VS) .
eq parseExitParam('any_['token[T]], M, VS) = 'any_['S[T]] .

eq parseDataExp('expBubble[T], M, VS) =
   if VS =/= mt then
      varsMaudeToLotos(meta-parse(
        addVarDeclSet(varsToVarDeclSet(VS), M), downQidList(T)))
   else meta-parse(M, downQidList(T))
   fi   .

eq parseAction('token[T], M, VS) = < 'G[T], mt > .
eq parseAction({'i}'SimpleAction, M, VS) = < {'i}'SimpleAction, mt > .
```

```
  eq parseAction('__[T, T'], M, VS) =
      < '__[term(parseAction(T, M, VS)),
            term(parseOffer(T',M, VS))],
        vars(parseOffer(T',M, VS)) > .
  eq parseAction('_`[_`][T,T'], M, VS) =
      < '_`[_`][term(parseAction(T,M,VS)),
                parseDataExp(T',M,VS vars(parseAction(T,M,VS)))],
        vars(parseAction(T,M,VS)) > .

  eq parseOffer('!_[T], M, VS) = < '!_[parseDataExp(T,M,VS)] , mt > .
  eq parseOffer('?_['_:_['token[T],'token[T']]], M, VS) =
      < '?_['_:_['V[T], 'S[T']]], downQid(T) : downQid(T') > .

  eq parseProcess({'stop}'BehaviourExp, M, VS) = {'stop}'BehaviourExp .
  eq parseProcess({'exit}'BehaviourExp, M, VS) = {'exit}'BehaviourExp .
  eq parseProcess('exit`(_`)[T], M, VS) = 'exit`(_`)[parseExitParam(T,M, VS)] .
  eq parseProcess('_;_[T,T'],M,VS) =
      '_;_[term(parseAction(T,M,VS)),
           parseProcess(T',M, VS vars(parseAction(T,M,VS)))] .
  eq parseProcess('_`[`]_[T,T'], M, VS) =
      '_`[`]_[parseProcess(T,M, VS), parseProcess(T',M, VS)] .
  eq parseProcess('_|`[_`]|_[T,T',T''], M, VS) =
      '_|`[_`]|_[parseProcess(T,M, VS), parseGateIdList(T'), parseProcess(T'',M, VS)] .
  eq parseProcess('_||_[T,T'], M, VS) =
      '_||_[parseProcess(T,M, VS), parseProcess(T',M, VS)] .
  eq parseProcess('_|||_[T,T'], M,VS) =
      '_|||_[parseProcess(T,M, VS), parseProcess(T',M, VS)] .
  eq parseProcess('_`[>_[T,T'], M,VS) =
      '_`[>_[parseProcess(T,M, VS), parseProcess(T',M, VS)] .
  eq parseProcess('hide_in_[T,T'], M, VS) =
      'hide_in_[parseGateIdList(T), parseProcess(T',M, VS)] .
  eq parseProcess('`[_`]->_[T,T'], M, VS) =
      '`[_`]->_[parseDataExp(T,M,VS), parseProcess(T',M,VS)] .

  eq parseGateIdList('token[T]) = 'G[T] .
  eq parseGateIdList('_`,_[T,T']) = '_`,_[parseGateIdList(T),
                                          parseGateIdList(T')] .

endfm
```

3.4 LOTOS command processing

The following module contains auxiliary functions that are used to process the tool commands. They will be used from the rules that define the database handling (see Section 3.5).

```
fmod LOTOS-COMMAND-PROCESSING is
  pr META-LEVEL .
  pr UNIT-DECL-PARSING .
  pr DECL-META-PRETTY-PRINT .

  op meta-pretty-print-transitions : Module Term -> QidList .
  op meta-pretty-print-transitions : Module TermList MachineInt -> QidList .
  op meta-pretty-print-trace : Module Term -> QidList .
  op meta-pretty-print-condition : Module Term -> QidList .

  vars QI F C S V Q Q1 Q2 : Qid .
  var QIL : QidList .
  vars N : MachineInt .
  var M : Module .
  vars T T' T'' PS N' T1 T2 T3 : Term .
  var TL : TermList .
```

```
op filter : QidList -> QidList .
op filter2 : QidList -> QidList .

eq filter(nil) = nil .
eq filter(Q QIL) = if Q == 'V or Q == 'S or Q == 'G then
                     filter2(QIL)
                   else (Q filter(QIL)) fi .
eq filter2(Q1 Q Q2 QIL) = strip(Q) filter(QIL) .

eq meta-pretty-print-transitions(M, T) =
  if T == {'nil}'PairSeq then
     ('\n 'No 'more 'transitions '. '\n)
  else
    ('\n 'TRANSITIONS ': '\n
     meta-pretty-print-transitions(M, T, 1) '\n )
  fi .

eq meta-pretty-print-transitions(M, {'nil}'PairSeq, N) = nil .
eq meta-pretty-print-transitions(M, '<_`,_> [ T, N' ], N) =
     '\n conc(index(' , N), '.) filter(my-meta-pretty-print(M,downTerm(T))) .
eq meta-pretty-print-transitions(M, '_|_ [ '<_`,_> [ T, N' ], TL], N) =
     '\n conc(index(' , N), '.) filter(my-meta-pretty-print(M,downTerm(T)))
     meta-pretty-print-transitions(M, TL, N + 1) .
eq meta-pretty-print-transitions(M, (T, TL), N) =
     meta-pretty-print-transitions(M, T, N)
     meta-pretty-print-transitions(M, TL, N + 1) .

eq meta-pretty-print-trace(M, T) =
     ('\n 'Trace ': filter(my-meta-pretty-print(M,T)) '\n) .

eq meta-pretty-print-condition(M, T) =
     ('\n 'Condition ': filter(my-meta-pretty-print(M,T)) '\n) .


op selectSol : MachineInt TermList -> Term .

eq selectSol(1, '<_`,_> [ T, N' ]) = '<_`,_> [ T, N' ] .
eq selectSol(1, '_|_ [ '<_`,_> [ T, N' ], TL]) = '<_`,_> [ T, N' ] .
ceq selectSol(N, '_|_ [ '<_`,_> [ T, N' ], TL]) =
   selectSol(_-_(N,1) , TL) if N > 1 .
eq selectSol(1, (T,TL)) = T .
ceq selectSol(N, (T,TL)) = selectSol(_-_(N,1), TL) if N > 1 .


op selectCond : Term -> Term .
op selectCond2 : Term -> Term .
op selectEvent : Term -> Term .
op selectEvent2 : Term -> Term .
op selectProc : Term -> Term .
op selectProc2 : Term -> Term .
op selectMaxNum : Term -> Term .
```

These functions receive as argument a metarepresented `PairSeq` having as components a meta-metarepresented possible transition and a metarepresented integer. They return a metarepresented condition, event, process, or integer.

```
eq selectCond( '<_`,_> [T, N'] ) = selectCond2(downTerm(T)) .
eq selectCond2('|--_--_-->_ [T1, T2, T3]) = T1 .

eq selectEvent( '<_`,_> [T, N'] ) = selectEvent2(downTerm(T)) .
```

```
  eq selectEvent2('|--_--_-->_ [T1, T2, T3]) = T2 .

  eq selectProc( '<_‘,_> [T, N'] ) = selectProc2(downTerm(T)) .
  eq selectProc2('|--_--_-->_ [T1, T2, T3]) = T3 .

  eq selectMaxNum( '<_‘,_> [T, N'] ) = N' .

endfm
```

3.5 Extending the Database by Inheritance

In [7] it is explained how the persistent state of the Full Maude system is given by a single object (of class `DatabaseClass`), which maintains the database of the system.

We can extend the Full Maude system by defining subclasses of `DatabaseClass` inheriting its behaviour and adding new functionalities to it, new attributes, etc.

```
mod LOTOS-DATABASE-HANDLING is
  pr EXT-DATABASE-HANDLING .
  pr META-LOTOS-TOOL-SIGN .
  pr LOTOS-COMMAND-PROCESSING .
  pr ACTONE-TRANSLATION .
  pr LOTOS-PARSING .

  sort Lotos-DB .
  subsort Lotos-DB < DatabaseClass .

  op Lotos-DB : -> Lotos-DB .
```

We define attributes to maintain the LOTOS process we are working with, the set of possible transitions of this process (computed after it has been asked with the corresponding command), and the trace of events and the conjunction of transition conditions of transitions already executed.

```
  op lotosProcess :_ : Term -> Attribute .
  op transitions :_ : Term -> Attribute .

  op trace :_ : Term -> Attribute .
  op condition :_ : Term -> Attribute .

  var Atts : AttributeSet .
  var X@Lotos-DB : Lotos-DB .
  var O : Oid .

  vars T T' T'' T''' T1 T2 T3 : Term .
  var DB : Database .
  var X@Database : DatabaseClass .
  var MN : ModName .
```

The following rules describes the behaviour associated with the new commands.

Rule `spec` says that if a specification is in the input, then the database of modules has to be modified by introducing modules corresponding to the data types (by means of operation `translate`), a module `EXT-LOTOS-SYNTAX` that includes the data types and the LOTOS syntax, a module `EXT-LOTOS-OK` that includes the data types and the LOTOS symbolic semantics, and a module `FULL-LOTOS` that includes the instantiation of the search strategy and defines the constant `MOD` to be equal to the metarepresentation of module `EXT-LOTOS-OK`. The `lotosProcess` attribute is also set to the process introduced in the specification (after being parsed with `parseProcess`).

```
  rl [spec] :
     < O : X@Lotos-DB |
```

```
      input : ('specification__behaviour_endspec[T,T', T'']),
      output : nil,
      db : DB, default : MN, lotosProcess : T''', Atts >
   => < O : X@Lotos-DB | input : nilTermList,
        output : ('\n 'Introduced 'specification parseModName(T)),
        db :
        processUnit(meta-parse(EXT-GRAMMAR,
               'fmod 'FULL-LOTOS 'is
                  'including 'LOTOS-TOOLS '.
                  'eq 'MOD '= 'up ''( 'EXT-LOTOS-OK '') '.
               'endfm),
             evalUnit(
               mod 'EXT-LOTOS-OK is
                 nilParameterList
                 including parseModName(T) .
                 including 'LOTOS-OK .
                 sorts none .
                 none none none none none none endm,
             evalUnit(
               fmod 'EXT-LOTOS-SYNTAX is
                 nilParameterList
                 including parseModName(T) .
                 including 'LOTOS-SYNTAX .
                 sorts none .
                 none none none none none endfm,
             translate(T, T', DB)))),
        default : parseModName(T),
        lotosProcess : parseProcess(T'',
          getFlatUnit(parseModName(T), translate(T, T', DB)),mt), Atts >  .

rl [show] :
   < O : X@Lotos-DB |
      input : ({'L`show`process`.}'LotosCommand),
      output : nil, db : DB,
      lotosProcess : T, Atts >
   => < O : X@Lotos-DB | input : nilTermList,
        output : ('This 'is 'a 'LOTOS 'process '. '\n
                      filter(meta-pretty-print(getFlatUnit('EXT-LOTOS-SYNTAX, DB), T))),
        db : DB, lotosProcess : T, Atts >  .

rl [show-transitions] :
   < O : X@Lotos-DB |
      input : ('L`show`possible`transitions`of_.[T]), Atts >
   => < O : X@Lotos-DB |
        input : ('L`trans`of_.[T]), Atts > .

rl [show-transitions] :
   < O : X@Lotos-DB | db : DB,
      input : ('L`trans`of_.[T]),
      output : nil,
      default : MN,
      lotosProcess : T', Atts >
   => < O : X@Lotos-DB | db : DB,
      input : ({'L`trans`.}'LotosCommand),
      output : nil, default : MN,
      lotosProcess : parseProcess(T,getFlatUnit(MN, DB),mt), Atts > .


rl [show-transitions] :
   < O : X@Lotos-DB | db : DB,
      input : ({'L`trans`.}'LotosCommand),
```

```
      output : nil,
      lotosProcess : T', transitions : T'', trace : T1, condition : T2, Atts >
   => if meta-reduce(getFlatUnit('FULL-LOTOS, DB),
                  'transitions[up(T')] ) =/= error*
      then
    < O : X@Lotos-DB | db : DB,
         input : nilTermList,
         output : (meta-pretty-print-transitions(getFlatUnit('EXT-LOTOS-OK, DB),
                    meta-reduce(getFlatUnit('FULL-LOTOS, DB),
                        'transitions[up(T')] ))),
         lotosProcess : T',
         transitions : meta-reduce(getFlatUnit('FULL-LOTOS, DB),
                  'transitions[up(T')] ),
         trace : ({'nil}'Trace),
         condition : ({'true}'TransCond),
         Atts >
      else
    < O : X@Lotos-DB | db : DB,
         input : nilTermList,
         output : ('ERROR 'in 'function 'transitions ),
         lotosProcess : T', transitions : T'', trace : T1, condition : T2, Atts >
      fi .


rl [show-transitions] :
   < O : X@Lotos-DB |
      input : ({'L`show`possible`transitions`.}'LotosCommand), Atts >
   => < O : X@Lotos-DB |
         input : ({'L`trans`.}'LotosCommand), Atts > .


rl [continue] :
   < O : X@Lotos-DB |
      input : ({'L`cont`.}'LotosCommand), Atts >
   => < O : X@Lotos-DB |
         input : ('L`cont_.[up(1)]), Atts > .

rl [continue] :
   < O : X@Lotos-DB | db : DB,
      input : ('L`cont_.[T]),
      output : nil,
      lotosProcess : T', transitions : T'', trace : T1, condition : T2, Atts >
   => if meta-reduce(getFlatUnit('FULL-LOTOS, DB),
                  'transitions[up(selectProc(selectSol(downMachineInt(T),T''))),
                         selectMaxNum(selectSol(downMachineInt(T),T''))] ) =/= error*
      then
    < O : X@Lotos-DB | db : DB,
         input : ({'L`show`state`.}'LotosCommand),
         output : nil,
         lotosProcess : T',
         transitions : meta-reduce(getFlatUnit('FULL-LOTOS, DB),
                  'transitions[up(selectProc(selectSol(downMachineInt(T),T''))),
                         selectMaxNum(selectSol(downMachineInt(T),T''))]),
         trace : meta-reduce(getFlatUnit('EXT-LOTOS-OK, DB),
           '__[T1, selectEvent(selectSol(downMachineInt(T),T''))]),
         condition : meta-reduce(getFlatUnit('EXT-LOTOS-OK, DB),
           '_/\_[T2, selectCond(selectSol(downMachineInt(T),T''))]),
         Atts >
      else
    < O : X@Lotos-DB | db : DB,
         input : nilTermList,
```

```
          output : ('ERROR ),
          lotosProcess : T', transitions : T'', trace : T1, condition : T2, Atts >
       fi .

  rl [show-state] :
     < O : X@Lotos-DB | db : DB,
       input : ({'L`show`state`.}'LotosCommand),
       output : nil,
       lotosProcess : T', transitions : T'', trace : T1, condition : T2, Atts >
     => < O : X@Lotos-DB | db : DB,
       input : nilTermList,
       output :
          (meta-pretty-print-trace(getFlatUnit('EXT-LOTOS-OK, DB), T1)
           meta-pretty-print-condition(getFlatUnit('EXT-LOTOS-OK, DB), T2)
           meta-pretty-print-transitions(getFlatUnit('EXT-LOTOS-OK, DB), T'')),
       lotosProcess : T',
       transitions : T'', trace : T1, condition : T2, Atts > .

endm
```

3.6 The Full Maude Environment of the LOTOS tool

Finally, we give the rules to initialize the loop, and to specify the communication between the loop (the input/output of the system) and the persistent state of the system. The following module is a redefinition of module `FULL-MAUDE` presented in [7], and it is prepared to receive both Full Maude input and the LOTOS tool input.

```
mod LOTOS-TOOL&FULL-MAUDE is

  pr META-LOTOS-TOOL-SIGN .
  pr LOTOS-DATABASE-HANDLING .
  pr PREDEF-UNITS .
  inc LOOP-MODE .

  subsort Object < State .

  op o : -> Oid .
  op init : -> System .

  var Atts : AttributeSet .
  var X@Lotos-DB : Lotos-DB .
  var X@Database : DatabaseClass .
  var O : Oid .
  var Q : Qid .
  vars QIL QIL' QIL'' : QidList .
```

The `init` rule initializes the persistent object as an object of class `Lotos-DB` by initializing its attributes.

```
  rl [init] :
     init
     => [nil,
         < o : Lotos-DB |
            db : evalUnit(CONFIGURATION,
                   evalUnit(TRIV, evalUnit(UP, emptyDatabase))),
            input : nilTermList, output : nil,
            default : 'META-LEVEL,
            lotosProcess : error*,
            transitions : error*,
            condition : ({'true}'TransCond),
```

```
          trace : ({'nil}'Trace) >,
        ('\n '\t  'Full 'Maude 'version '1.0.5
         '\n '\t  'Copyright '1999-2000 'SRI 'International
         '\n '&
         '\n '\t  'LOTOS 'in 'Maude 'version '0.1
         '\n )] .
```

The `in` rules use the `EXT-GRAMMAR` to parse Full Maude modules and commands, and the `LOTOS-GRAMMAR` to parse Full LOTOS specifications and commands.

```
  crl [in] :
     [Q QIL,
      < O : X@Lotos-DB | input : nilTermList,
                         output : nil, Atts >,
      QIL']
     => if meta-parse(EXT-GRAMMAR, Q QIL) == error*
        then [nil,
                 < O : X@Lotos-DB |
                 input : nilTermList,
                 output : ('ERROR: 'incorrect 'input '.), Atts >,
              QIL']
        else [nil,
              < O : X@Lotos-DB |
                 input : meta-parse(EXT-GRAMMAR, Q QIL),
                 output : nil, Atts >,
              QIL']
        fi
        if Q =/= 'specification and Q =/= 'L .

  crl [in] :
     [Q QIL,
      < O : X@Lotos-DB | input : nilTermList,
                         output : nil, Atts >,
      QIL']
     => if meta-parse(LOTOS-GRAMMAR, Q QIL) == error*
           then [nil,
                 < O : X@Lotos-DB |
                 input : nilTermList,
                 output : ('ERROR: 'incorrect 'LOTOS 'TOOL 'input '.), Atts >,
              QIL']
              else [nil,
                 < O : X@Lotos-DB |
                 input : meta-parse(LOTOS-GRAMMAR, Q QIL),
                 output : nil, Atts >,
              QIL']
              fi
        if Q == 'specification or Q == 'L .
```

The `out` rule, dealing with output to the Maude system, is left unmodified.

```
  crl [out] :
     [QIL,
      < O : X@Database | output : QIL', Atts >,
      QIL'']
     => [QIL,
         < O : X@Database | output : nil, Atts >,
         (QIL' QIL'')]
        if QIL' =/= nil .

endm
```

After introducing this last module into the Maude system, the tool is able to receive Maude modules. We can introduce the modules defined in Section 2, in order to have them in the database of modules, and be able to use them when needed. At that moment the LOTOS tool is also usable, and LOTOS specifications can be entered and executed.

3.7 Execution example

This is an example of an interaction with the LOTOS tool.

```
Maude> (specification SPEC
type NAT is
  sorts NAT
  opns
    zero : -> NAT
    succ : NAT -> NAT
    add : NAT , NAT -> NAT
    eq : NAT , NAT -> Bool
  eqns
    forall N : NAT,
           M : NAT
    ofsort NAT
      add(zero, N) = N ;
      add(succ(N), M) = succ(add(N,M)) ;
    ofsort Bool
      eq(zero, zero) = true ;
      eq(zero, succ(N)) = false ;
      eq(succ(N), zero) = false ;
      eq(succ(N), succ(M)) = eq(N,M) ;
endtype
behaviour
 h ! zero ; stop
[]
(
   g ! (succ(zero)) ; stop
 |[ g ]|
   g ? x : NAT [ eq(x,succ(zero)) ] ; h ! (add(x, succ(zero))) ; stop
)
endspec)

Introduced specification SPEC
Maude> (L trans .)

TRANSITIONS :

1. |-- eq ( x , succ ( zero ) ) /\ x = succ ( zero ) -- g succ ( zero ) -->
    stop | [ g ] | h ! succ ( succ ( zero ) ) ; stop
2. |-- true -- h zero --> stop

Maude> (L cont 1 .)

Trace : g succ ( zero )

Condition : eq ( x , succ ( zero ) ) /\ x = succ ( zero )

TRANSITIONS :

1. |-- true -- h succ ( succ ( zero ) ) --> stop | [ g ] | stop

Maude> (L cont .)
```

```
Trace : ( g succ ( zero ) ) ( h succ ( succ ( zero ) ) )

Condition : eq ( x , succ ( zero ) ) /\ x = succ ( zero )

No more transitions .
```

4 Conclusions and future work

We have presented a new example of how rewriting logic and Maude can be used as a semantic framework and metalanguage, where entire environments and tools for the execution of formal specification languages can be built. In this process, reflection plays a decisive role.

In [19] we compare our approach (applied to the CCS specification language) with the logical framework and theorem prover Isabelle [16]. We conclude that by using rewriting logic we can use higher-order techniques in a first-order framework by means of reflection; that is, reflection provides a first-order system like Maude with most of the power of a higher-order system like Isabelle.

In this paper we have shown how new functionalities, such as parsing, pretty printing and input/output processing, can be added to a specific formal tool by using Maude.

Based on the symbolic semantics used in this paper, a symbolic bisimulation [3] and a modal logic [1] have been defined. We plan to extend our tool so we can check if two processes are bisimilar, or if a process fulfills a given modal logic formula.

Acknowledgements

I would like to thank Carron Shankland for her helpful answers about LOTOS symbolic semantics. I am also very grateful to Narciso Martí-Oliet for his remarks on earlier versions of this paper.

References

[1] M. Calder, S. Maharaj, and C. Shankland. An adequate logic for Full LOTOS. In J. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 384–395. Springer, 2001.

[2] M. Calder and C. Shankland. A symbolic semantics and bisimulation for Full LOTOS. Technical Report CSM-159, University of Stirling, 2000.

[3] M. Calder and C. Shankland. A symbolic semantics and bisimulation for Full LOTOS. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proceedings of FORTE 2001, 21st International Conference on Formal Techniques for Networked and Distributed Systems*, pages 184–200. Kluwer Academic Publishers, 2001.

[4] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude as a metalanguage. In C. Kirchner and H. Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International, 1999. `http://maude.cs.uiuc.edu/maude1/manual`.

[7] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, 1999.

[8] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer, 1985.

[9] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[11] ISO/IEC. *LOTOS—A formal description technique based on the temporal ordering of observational behaviour*. International Standard 8807, International Organization for standardization — Information Processing Systems — Open Systems Interconnection, Geneva, 1989.

[12] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, 1993. `http://maude.cs.uiuc.edu/papers`.

[13] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[14] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, NATO ASI Series F: Computer and Systems Sciences 165, pages 347–398. Springer, 1998.

[15] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[16] L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[17] A. Verdejo and N. Martí-Oliet. Executing and verifying CCS in Maude. Technical Report 99-00, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2000.

[18] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In T. Bolognesi and D. Latella, editors, *Formal Methods For Distributed System Development. FORTE/PSTV 2000 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX) October 10–13, 2000, Pisa, Italy*, pages 351–366. Kluwer Academic Publishers, 2000.

[19] A. Verdejo and N. Martí-Oliet. Executing and verifying CCS in Maude. Manuscript submitted for publication, 2001.

A ACT ONE and LOTOS signatures

These are the grammars used to parse ACT ONE specifications and Full LOTOS behaviour expressions.

```
fmod ACTONE-SIGN is

  sort Token NeTokenList Bubble ExpBubble .

  sorts OperDecl OperDeclList Decl DeclList TypeDecl TypeDeclList
        SortName SortNameList EqDecl EqDeclList EqDeclGroup VariDecl
        VariDeclList VariDeclGroup VarEqDeclList .

  subsort TypeDecl < TypeDeclList .
  op __ : TypeDeclList TypeDeclList -> TypeDeclList
            [assoc prec 25 gather(e E)] .
```

```
  subsort VariDecl < VariDeclList .
  op _,_ : VariDeclList VariDeclList -> VariDeclList  [assoc prec 14] .

  subsort EqDeclGroup VariDeclGroup < VarEqDeclList .
  op __ : VarEqDeclList VarEqDeclList -> VarEqDeclList [assoc prec 16] .


  subsort EqDecl < EqDeclList .
  op __ : EqDeclList EqDeclList -> EqDeclList [assoc prec 7] .

  subsort Token < SortName .

  sort TokenList .
  subsort Token < TokenList .
  op _,_ : TokenList TokenList -> TokenList [assoc prec 4] .

  subsort TokenList < SortNameList .


  subsort Decl < DeclList .
  subsort OperDecl < OperDeclList .
  op __ : OperDeclList OperDeclList -> OperDeclList [assoc prec 7] .

  op __ : DeclList DeclList -> DeclList [assoc prec 19 gather(e E)] .

  op _:_ : TokenList Token -> VariDecl [prec 10] .

  op _: ->_ : Token SortName -> OperDecl [prec 5] .
  op _:_->_ : Token SortNameList SortName -> OperDecl [prec 5] .

  op _=_; : Bubble Bubble -> EqDecl [prec 5] .
  op _=>_=_; : Bubble Bubble Bubble -> EqDecl [prec 5] .

  op forall_ : VariDeclList -> VariDeclGroup [prec 15 gather(e)] .
  op ofsort__ : Token EqDeclList -> EqDeclGroup [prec 15] .

  op sorts_ : Token -> Decl [prec 18 gather(e)] .
  op opns_ : OperDeclList -> Decl [prec 18 gather(e)] .
  op eqns_ : VarEqDeclList -> Decl [prec 18 gather(e)] .


  op type_is_endtype : Token DeclList -> TypeDecl  [prec 20 gather(e e)] .
  op type_is__endtype : Token TokenList DeclList -> TypeDecl
       [prec 20 gather(e e e)] .

endfm

fmod LOTOS-SIGN is
  pr ACTONE-SIGN .

  *** identifiers contructors
  sorts VarId SortId GateId .

  subsorts Token < VarId SortId GateId .

  sort DataExp .

  subsort ExpBubble < DataExp .

  sort BehaviourExp .
```

```
  op stop : -> BehaviourExp .
  op exit : -> BehaviourExp .
  op exit`(_`) : ExitParam -> BehaviourExp .

  sort ExitParam .
  subsort DataExp < ExitParam .

  op any_ : SortId -> ExitParam .

  sorts SimpleAction StrucAction Action Offer SelecPred IdDecl .

  subsort GateId < SimpleAction .
  subsorts SimpleAction StrucAction < Action .

  op i : -> SimpleAction .
  op __ : GateId Offer -> StrucAction [prec 30] .

  op !_ : DataExp -> Offer [prec 25 gather(e)]  .
  op ?_ : IdDecl -> Offer [prec 25] .
  op _:_ : VarId SortId -> IdDecl [prec 20] .

  op _`[_`] : SimpleAction SelecPred -> Action [prec 30] .
  op _`[_`] : StrucAction SelecPred -> Action [prec 30] .

  subsort ExpBubble < SelecPred .

  *** action prefixing
  op _;_ : Action BehaviourExp -> BehaviourExp [prec 35] .

  *** choice
  op _`[`]_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .
  op choice_`[`]_ : IdDecl BehaviourExp -> BehaviourExp [prec 40] .

  *** parallelism
  sort GateIdList .
  subsort GateId < GateIdList .

  subsort TokenList < GateIdList .

  op _|`[_`]|_ : BehaviourExp GateIdList BehaviourExp ->
                 BehaviourExp [prec 40] .

  op _||_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .
  op _|||_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .

  *** disable
  op _`[>_ : BehaviourExp BehaviourExp -> BehaviourExp [prec 40] .

  *** guard
  op `[_`]->_ : SelecPred BehaviourExp -> BehaviourExp [prec 40] .

  *** hide
  op hide_in_ : GateIdList BehaviourExp -> BehaviourExp [prec 40] .

  sort Specification .

  op specification__behaviour_endspec :
    Token TypeDeclList BehaviourExp -> Specification [prec 50 gather(e e e)] .

endfm
```